How long, O Bayesian network, will I sample thee?

A program analysis perspective on expected sampling times

Kevin Batz(✉), Benjamin Lucien Kaminski(✉), Joost-Pieter Katoen(✉), and Christoph Matheja(✉)

RWTH Aachen University, Aachen, Germany
kevin.batz@rwth-aachen.de,
{benjamin.kaminski,katoen,matheja}@cs.rwth-aachen.de

Abstract. Bayesian networks (BNs) are probabilistic graphical models for describing complex joint probability distributions. The main problem for BNs is inference: Determine the probability of an event given observed evidence. Since exact inference is often infeasible for large BNs, popular approximate inference methods rely on sampling.

We study the problem of determining the expected time to obtain a single valid sample from a BN. To this end, we translate the BN together with observations into a probabilistic program. We provide proof rules that yield the exact expected runtime of this program in a fully automated fashion. We implemented our approach and successfully analyzed various real–world BNs taken from the Bayesian network repository.

Keywords: Probabilistic programs · Expected runtimes
Weakest preconditions · Program verification

1 Introduction

Bayesian networks (BNs) are *probabilistic graphical models* representing joint probability distributions of sets of random variables with conditional dependencies. Graphical models are a popular and appealing modeling formalism, as they allow to succinctly represent complex distributions in a human–readable way. BNs have been intensively studied at least since 1985 [43] and have a wide range of applications including machine learning [24], speech recognition [50], sports betting [11], gene regulatory networks [18], diagnosis of diseases [27], and finance [39].

Probabilistic programs are programs with the key ability to draw values at random. Seminal papers by Kozen from the 1980s consider formal semantics [32] as well as initial work on verification [33,47]. McIver and Morgan [35] build on this work to further weakest–precondition style verification for imperative probabilistic programs.

© The Author(s) 2018
A. Ahmed (Ed.): ESOP 2018, LNCS 10801, pp. 186–213, 2018.
https://doi.org/10.1007/978-3-319-89884-1_7

The interest in probabilistic programs has been rapidly growing in recent years [20,23]. Part of the reason for this déjà vu is their use for representing probabilistic graphical models [31] such as BNs. The full potential of modern probabilistic programming languages like Anglican [48], Church [21], Figaro [44], R2 [40], or Tabular [22] is that they enable rapid prototyping and obviate the need to manually provide inference methods tailored to an individual model.

Probabilistic inference is the problem of determining the probability of an event given observed evidence. It is a major problem for both BNs and probabilistic programs, and has been subject to intense investigations by both theoreticians and practitioners for more than three decades; see [31] for a survey. In particular, it has been shown that for probabilistic programs exact inference is highly undecidable [28], while for BNs both *exact inference* as well as *approximate inference* to an arbitrary precision are NP–hard [12,13]. In light of these complexity–theoretical hurdles, a popular way to analyze probabilistic graphical models as well as probabilistic programs is to gather a large number of independent and identically distributed (i.i.d. for short) samples and then do statistical reasoning on these samples. In fact, all of the aforementioned probabilistic programming languages support sampling based inference methods.

Rejection sampling is a fundamental approach to obtain valid samples from BNs with observed evidence. In a nutshell, this method first samples from the joint (unconditional) distribution of the BN. If the sample complies with all evidence, it is valid and accepted; otherwise it is rejected and one has to resample.

Apart from rejection sampling, there are more sophisticated sampling techniques, which mainly fall in two categories: Markov Chain Monte Carlo (MCMC) and importance sampling. But while MCMC requires heavy hand–tuning and suffers from slow convergence rates on real–world instances [31, Chapter 12.3], virtually all variants of importance sampling rely again on rejection sampling [31,49].

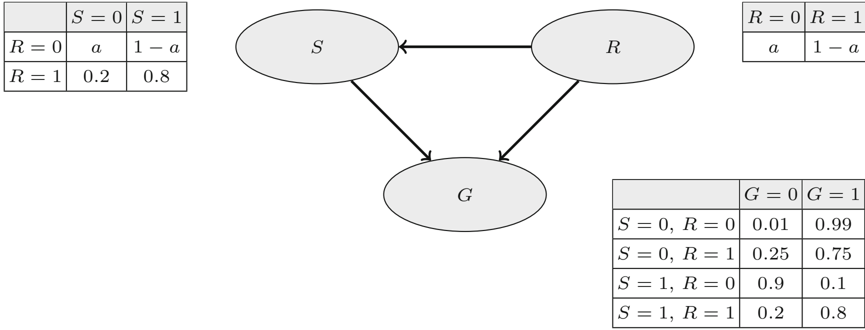
A major problem with rejection sampling is that for poorly conditioned data, this approach might have to reject and resample very often in order to obtain just a single accepting sample. Even worse, being poorly conditioned need not be immediately evident for a given BN, let alone a probabilistic program. In fact, Gordon et al. [23, p. 177] point out that

> “the main challenge in this setting [i.e. sampling based approaches] is that many samples that are generated during execution are ultimately rejected for not satisfying the observations.”

If too many samples are rejected, the expected sampling time grows so large that sampling becomes infeasible. The expected sampling time of a BN is therefore a key figure for deciding whether sampling based inference is the method of choice.

How Long, O Bayesian Network, will I Sample Thee? More precisely, we use techniques from program verification to give an answer to the following question:

> Given a Bayesian network with observed evidence, how long does it take in expectation to obtain a *single* sample that satisfies the observations?



	$S=0$	$S=1$
$R=0$	a	$1-a$
$R=1$	0.2	0.8

$R=0$	$R=1$
a	$1-a$

	$G=0$	$G=1$
$S=0,\ R=0$	0.01	0.99
$S=0,\ R=1$	0.25	0.75
$S=1,\ R=0$	0.9	0.1
$S=1,\ R=1$	0.2	0.8

Fig. 1. A simple Bayesian network.

As an example, consider the BN in Fig. 1 which consists of just three nodes (random variables) that can each assume values 0 or 1. Each node X comes with a conditional probability table determining the probability of X assuming some value given the values of all nodes Y that X depends on (i.e. X has an incoming edge from Y), see [3, Appendix A.1] for detailed calculations. For instance, the probability that G assumes value 0, given that S and R are both assume 1, is 0.2. Note that this BN is paramterized by $a \in [0, 1]$.

Now, assume that our observed evidence is the event $G = 0$ and we apply rejection sampling to obtain *one* accepting sample from this BN. Then our approach will yield that a rejection sampling algorithm will, on average, require

$$\frac{200a^2 - 40a - 460}{89a^2 - 69a - 21}$$

guard evaluations, random assignments, etc. until it obtains a single sample that complies with the observation $G = 0$ (the underlying runtime model is discussed in detail in Sect. 3.3). By examination of this function, we see that for large ranges of values of a the BN is rather well–behaved: For $a \in [0.08,\ 0.78]$ the expected sampling time stays below 18. Above $a = 0.95$ the expected sampling time starts to grow rapidly up to 300.

While 300 is still moderate, we will see later that expected sampling times of real–world BNs can be much larger. For some BNs, the expected sampling time even exceeded 10^{18}, rendering sampling based methods infeasible. In this case, exact inference (despite NP–hardness) was a viable alternative (see Sect. 6).

Our Approach. We apply weakest precondition style reasoning a lá McIver and Morgan [35] and Kaminski et al. [30] to analyze both expected outcomes and *expected runtimes* (ERT) of a *syntactic fragment of* pGCL, which we call the *Bayesian Network Language* (BNL). Note that since BNL is a syntactic fragment of pGCL, every BNL program is a pGCL program but *not vice versa*. The main restriction of BNL is that (in contrast to pGCL) loops are of a special form that prohibits undesired data flow across multiple loop iterations. While this

restriction renders BNL incapable of, for instance, counting the number of loop iterations[1], BNL is expressive enough to encode Bayesian networks with observed evidence.

For BNL, we develop dedicated proof rules to determine *exact* expected values and the *exact* ERT of any BNL program, including loops, without any user–supplied data, such as invariants [30,35], ranking or metering functions [19], (super)martingales [8–10], etc.

As a central notion behind these rules, we introduce *f–i.i.d.–ness* of probabilistic loops, a concept closely related to stochastic independence, that allows us to *rule out undesired parts of the data flow across loop iterations*. Furthermore, we show how every BN with observations is translated into a BNLprogram, such that

(a) executing the BNL program corresponds to sampling from the *conditional* joint distribution given by the BN and observed data, and
(b) the ERT of the BNL program corresponds to the expected time until a sample that satisfies the observations is obtained from the BN.

As a consequence, exact expected sampling times of BNs can be inferred by means of weakest precondition reasoning in a fully automated fashion. This can be seen as a first step towards formally evaluating the quality of a plethora of different sampling methods (cf. [31,49]) on source code level.

Contributions. To summarize, our main contributions are as follows:

- We develop easy–to–apply proof rules to reason about expected outcomes and expected runtimes of probabilistic programs with f–i.i.d. loops.
- We study a syntactic fragment of probabilistic programs, the Bayesian network language (BNL), and show that our proof rules are applicable to every BNL program; expected runtimes of BNL programs can thus be inferred.
- We give a formal translation from Bayesian networks with observations to BNL programs; expected sampling times of BNs can thus be inferred.
- We implemented a prototype tool that automatically analyzes the expected sampling time of BNs with observations. An experimental evaluation on real–world BNs demonstrates that very large expected sampling times (in the magnitude of millions of years) can be inferred within less than a second; This provides practitioners the means to decide whether sampling based methods are appropriate for their models.

Outline. We discuss related work in Sect. 2. Syntax and semantics of the probabilistic programming language pGCL are presented in Sect. 3. Our proof rules are introduced in Sect. 4 and applied to BNs in Sect. 5. Section 6 reports on experimental results and Sect. 7 concludes.

[1] An example of a program that is *not* expressible in BNL is given in Example 1.

2 Related Work

While various techniques for formal reasoning about runtimes and expected outcomes of probabilistic programs have been developed, e.g. [6,7,17,25,38], none of them explicitly apply formal methods to reason about Bayesian networks on source code level. In the following, we focus on approaches close to our work.

Weakest Preexpectation Calculus. Our approach builds upon the expected runtime calculus [30], which is itself based on work by Kozen [32,33] and McIver and Morgan [35]. In contrast to [30], we develop specialized proof rules for a clearly specified program fragment *without* requiring user–supplied invariants. Since finding invariants often requires heavy calculations, our proof rules contribute towards simplifying and automating verification of probabilistic programs.

Ranking Supermartingales. Reasoning about almost–sure termination is often based on ranking (super)martingales (cf. [8,10]). In particular, Chatterjee et al. [9] consider the class of affine probabilistic programs for which linear ranking supermartingales exist (Lrapp); thus proving (positive[2]) almost–sure termination for all programs within this class. They also present a doubly–exponential algorithm to approximate ERTs of Lrapp programs. While all BNL programs lie within Lrapp, our proof rules yield *exact* ERTs as *expectations* (thus allowing for compositional proofs), in contrast to a single number for a fixed initial state.

Bayesian Networks and Probabilistic Programs. Bayesian networks are a—if not the most—popular probabilistic graphical model (cf. [4,31] for details) for reasoning about conditional probabilities. They are closely tied to (a fragment of) probabilistic programs. For example, Infer.NET [36] performs inference by compiling a probabilistic program into a Bayesian network. While correspondences between probabilistic graphical models, such as BNs, have been considered in the literature [21,23,37], we are not aware of a formal soudness proof for a translation from classical BNs into probabilistic programs including conditioning.

Conversely, some probabilistic programming languages such as Church [21], Stan [26], and R2 [40] directly perform inference on the program level using sampling techniques similar to those developed for Bayesian networks. Our approach is a step towards understanding sampling based approaches formally: We obtain the exact expected runtime required to generate a sample that satisfies all observations. This may ultimately be used to evaluate the quality of a plethora of proposed sampling methods for Bayesian inference (cf. [31,49]).

3 Probabilistic Programs

We briefly present the probabilistic programming language that is used throughout this paper. Since our approach is embedded into weakest-precondition style approaches, we also recap calculi for reasoning about both expected outcomes and expected runtimes of probabilistic programs.

[2] Positive almost–sure termination means termination in finite expected time [5].

3.1 The Probabilistic Guarded Command Language

We enhance Dijkstra's Guarded Command Language [14,15] by a probabilistic construct, namely a random assignment. We thereby obtain a *probabilistic Guarded Command Language* (for a closely related language, see [35]).

Let Vars be a finite set of *program variables*. Moreover, let $\mathbb{Q}$ be the set of rational numbers, and let $\mathcal{D}(\mathbb{Q})$ be the set of discrete probability distributions over $\mathbb{Q}$. The set of *program states* is given by $\Sigma = \{\, \sigma \mid \sigma \colon \mathsf{Vars} \to \mathbb{Q} \,\}$.

A *distribution expression* μ is a function of type $\mu \colon \Sigma \to \mathcal{D}(\mathbb{Q})$ that takes a program state and maps it to a probability distribution on values from $\mathbb{Q}$. We denote by μ_σ the distribution obtained from applying σ to μ.

The probabilistic guarded command language (pGCL) is given by the grammar

$$
\begin{array}{lllr}
C & \longrightarrow & \texttt{skip} & \text{(effectless program)}\\
 & & \mid\ \texttt{diverge} & \text{(endless loop)}\\
 & & \mid\ x :\approx \mu & \text{(random assignment)}\\
 & & \mid\ C;\, C & \text{(sequential composition)}\\
 & & \mid\ \texttt{if}\,(\varphi)\,\{C\}\,\texttt{else}\,\{C\} & \text{(conditional choice)}\\
 & & \mid\ \texttt{while}\,(\varphi)\,\{C\} & \text{(while loop)}\\
 & & \mid\ \texttt{repeat}\,\{C\}\,\texttt{until}\,(\varphi)\ , & \text{(repeat–until loop)}
\end{array}
$$

where $x \in \mathsf{Vars}$ is a program variable, μ is a distribution expression, and φ is a Boolean expression guarding a choice or a loop. A pGCL program that contains neither `diverge`, nor `while`, nor `repeat` – `until` loops is called loop–free.

For $\sigma \in \Sigma$ and an arithmetical expression E over Vars, we denote by $\sigma(E)$ the evaluation of E in σ, i.e. the value that is obtained by evaluating E after replacing any occurrence of any program variable x in E by the value $\sigma(x)$. Analogously, we denote by $\sigma(\varphi)$ the evaluation of a guard φ in state σ to either true or false. Furthermore, for a value $v \in \mathbb{Q}$ we write $\sigma[x \mapsto v]$ to indicate that we set program variable x to value v in program state σ, i.e.[3]

$$
\sigma[x \mapsto v] \;=\; \lambda\, y \bullet \begin{cases} v, & \text{if } y = x\\ \sigma(y), & \text{if } y \neq x\ . \end{cases}
$$

We use the Iverson bracket notation to associate with each guard its according indicator function. Formally, the Iverson bracket $[\varphi]$ of φ is thus defined as the function $[\varphi] = \lambda\,\sigma \bullet\ \sigma(\varphi)$.

Let us briefly go over the pGCL constructs and their effects: `skip` does not alter the current program state. The program `diverge` is an infinite busy loop, thus takes infinite time to execute. It returns no final state whatsoever.

The random assignment $x :\approx \mu$ is (a) the only construct that can actually alter the program state and (b) the only construct that may introduce random

[3] We use λ–expressions to construct functions: Function $\lambda X \bullet\ \epsilon$ applied to an argument α evaluates to ϵ in which every occurrence of X is replaced by α.

behavior into the computation. It takes the current program state σ, then *samples* a value v from probability distribution μ_σ, and then assigns v to program variable x. An example of a random assignment is

$$x :\approx 1/2 \cdot \langle 5 \rangle + 1/6 \cdot \langle y+1 \rangle + 1/3 \cdot \langle y-1 \rangle \ .$$

If the current program state is σ, then the program state is altered to either $\sigma[x \mapsto 5]$ with probability $1/2$, or to $\sigma[x \mapsto \sigma(y)+1]$ with probability $1/6$, or to $\sigma[x \mapsto \sigma(y)-1]$ with probability $1/3$. The remainder of the pGCL constructs are standard programming language constructs.

In general, a pGCL program C is executed on an input state and yields a *probability distribution* over final states due to possibly occurring random assignments inside of C. We denote that resulting distribution by $[\![C]\!]_\sigma$. Strictly speaking, programs can yield *subdistributions*, i.e. probability distributions whose total mass may be below 1. The "missing"probability mass represents the probability of nontermination. Let us conclude our presentation of pGCLwith an example:

Example 1 (Geometric Loop). Consider the program C_{geo} given by

$$\begin{aligned}&x :\approx 0; \quad c :\approx 1/2 \cdot \langle 0 \rangle + 1/2 \cdot \langle 1 \rangle;\\ &\texttt{while}\,(c=1)\,\{x :\approx x+1;\; c :\approx 1/2 \cdot \langle 0 \rangle + 1/2 \cdot \langle 1 \rangle\}\end{aligned}$$

This program basically keeps flipping coins until it flips, say, heads ($c = 0$). In x it counts the number of unsuccessful trials.[4] In effect, it almost surely sets c to 0 and moreover it establishes a geometric distribution on x. The resulting distribution is given by

$$[\![C_{geo}]\!]_\sigma(\tau) \;=\; \sum_{n=0}^{\omega} [\tau = \sigma[c, x \mapsto 0, n]] \cdot \frac{1}{2^{n+1}} \ . \qquad \triangle$$

3.2 The Weakest Preexpectation Transformer

We now present the weakest preexpectation transformer wp for reasoning about expected outcomes of executing probabilistic programs in the style of McIver and Morgan [35]. Given a random variable f mapping program states to reals, it allows us to reason about the expected value of f after executing a probabilistic program on a given state.

Expectations. The random variables the wp transformer acts upon are taken from a set of so-called expectations, a term coined by McIver and Morgan [35]:

[4] This counting is also the reason that C_{geo} is an example of a program that is not expressible in our BNL language that we present later.

Definition 1 (Expectations). *The set of expectations* $\mathbb{E}$ *is defined as*

$$\mathbb{E} \;=\; \left\{ f \;\middle|\; f\colon \Sigma \to \mathbb{R}_{\geq 0}^{\infty} \right\} \;.$$

We will use the notation $f[x/E]$ *to indicate the* replacement *of every occurrence of* x *in* f *by* E*. Since* x*, however, does not actually occur in* f*, we more formally define* $f[x/E] = \lambda\sigma \bullet\; f(\sigma[x \mapsto \sigma(E)])$*.*

A complete partial order $\leq$ *on* $\mathbb{E}$ *is obtained by point–wise lifting the canonical total order on* $\mathbb{R}_{\geq 0}^{\infty}$*, i.e.*

$$f_1 \;\preceq\; f_2 \quad \text{iff} \quad \forall \sigma \in \Sigma\colon \quad f_1(\sigma) \;\leq\; f_2(\sigma) \;.$$

Its least element is given by $\lambda\sigma \bullet\; 0$ *which we (by slight abuse of notation) also denote by* 0*. Suprema are constructed pointwise, i.e. for* $S \subseteq \mathbb{E}$ *the supremum* $\sup S$ *is given by* $\sup S = \lambda\sigma \bullet\; \sup_{f \in S}\; f(\sigma)$*.*

We allow expectations to map only to positive reals, so that we have a complete partial order readily available, which would not be the case for expectations of type $\Sigma \to \mathbb{R} \cup \{-\infty, +\infty\}$. A wp calculus that *can* handle expectations of such type needs more technical machinery and cannot make use of this underlying natural partial order [29]. Since we want to reason about ERTs which are by nature non–negative, we will not need such complicated calculi.

Notice that we use a slightly different definition of expectations than McIver and Morgan [35], as we allow for *unbounded* expectations, whereas [35] requires that expectations are *bounded*. This however would prevent us from capturing ERTs, which are potentially unbounded.

Expectation Transformers. For reasoning about the expected value of $f \in \mathbb{E}$ after execution of C, we employ a backward–moving weakest preexpectation transformer $\mathsf{wp}[\![C]\!]\colon \mathbb{E} \to \mathbb{E}$, that maps a *postexpectation* $f \in \mathbb{E}$ to a *preexpectation* $\mathsf{wp}\,[\![C]\!]\,(f) \in \mathbb{E}$, such that $\mathsf{wp}\,[\![C]\!]\,(f)\,(\sigma)$ is the expected value of f after executing C on initial state σ. Formally, if C executed on input σ yields final distribution $[\![C]\!]_\sigma$, then the *weakest preexpectation* $\mathsf{wp}\,[\![C]\!]\,(f)$ *of* C *with respect to postexpectation* f *is given by*

$$\mathsf{wp}\,[\![C]\!]\,(f)\,(\sigma) \;=\; \int_{\Sigma} \; f \; d[\![C]\!]_\sigma \;, \tag{1}$$

where we denote by $\int_A \; h \; d\nu$ the expected value of a random variable $h\colon A \to \mathbb{R}_{\geq 0}^{\infty}$ with respect to a probability distribution $\nu\colon A \to [0,\, 1]$. Weakest preexpectations can be defined in a very systematic way:

Definition 2 (The wp Transformer [35]). *The weakest preexpectation transformer* $\mathsf{wp}\colon \mathsf{pGCL} \to \mathbb{E} \to \mathbb{E}$ *is defined by induction on all* pGCL *programs according to the rules in Table 1. We call* $F_f(X) = [\neg\varphi] \cdot f + [\varphi] \cdot \mathsf{wp}\,[\![C]\!]\,(X)$ *the* wp*–*characteristic functional *of the loop* $\mathtt{while}\,(\varphi)\,\{C\}$ *with respect to postexpectation* f*. For a given* wp*–characteristic function* F_f*, we call the sequence* $\{F_f^n(0)\}_{n \in \mathbb{N}}$ *the* orbit of F_f*.*

Table 1. Rules for the wp–transformer.

C	$\mathsf{wp}\,[\![C]\!]\,(f)$
`skip`	f
`diverge`	0
$x :\approx \mu$	$\lambda\sigma\bullet\; \int_{\mathbb{Q}} \big(\lambda v\bullet\; f[x/v]\big)\, d\mu_\sigma$
$\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$	$[\varphi]\cdot \mathsf{wp}\,[\![C_1]\!]\,(f) + [\neg\varphi]\cdot \mathsf{wp}\,[\![C_2]\!]\,(f)$
$C_1;\, C_2$	$\mathsf{wp}\,[\![C_1]\!]\,(\mathsf{wp}\,[\![C_2]\!]\,(f))$
$\texttt{while}\,(\varphi)\,\{C'\}$	$\mathsf{lfp}\; X\bullet\;\; [\neg\varphi]\cdot f + [\varphi]\cdot \mathsf{wp}\,[\![C']\!]\,(X)$
$\texttt{repeat}\,\{C'\}\,\texttt{until}\,(\varphi)$	$\mathsf{wp}\,[\![C';\, \texttt{while}\,(\neg\varphi)\,\{C'\}]\!]\,(f)$

Let us briefly go over the definitions in Table 1: For `skip` the program state is not altered and thus the expected value of f is just f. The program `diverge` will never yield any final state. The distribution over the final states yielded by `diverge` is thus the null distribution $\nu_0(\tau) = 0$, that assigns probability 0 to *every* state. Consequently, the expected value of f after execution of `diverge` is given by $\int_\Sigma\; f\; d\nu_0 = \sum_{\tau\in\Sigma} 0\cdot f(\tau) = 0$.

The rule for the random assignment $x :\approx \mu$ is a bit more technical: Let the current program state be σ. Then for every value $v \in \mathbb{Q}$, the random assignment assigns v to x with probability $\mu_\sigma(v)$, where σ is the current program state. The value of f after assigning v to x is $f(\sigma\,[x \mapsto v]) = f[x/v](\sigma)$ and therefore the expected value of f after executing the random assignment is given by

$$\sum_{v\in\mathbb{Q}} \mu_\sigma(v)\cdot f[x/v](\sigma) \;=\; \int_{\mathbb{Q}} \big(\lambda v\bullet\; f[x/v](\sigma)\big)\; d\mu_\sigma\;.$$

Expressed as a function of σ, the latter yields precisely the definition in Table 1.

The definition for the conditional choice $\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$ is not surprising: if the current state satisfies φ, we have to opt for the weakest preexpectation of C_1, whereas if it does not satisfy φ, we have to choose the weakest preexpectation of C_2. This yields precisely the definition in Table 1.

The definition for the sequential composition $C_1;\, C_2$ is also straightforward: We first determine $\mathsf{wp}\,[\![C_2]\!]\,(f)$ to obtain the expected value of f after executing C_2. Then we mentally prepend the program C_2 by C_1 and therefore determine the expected value of $\mathsf{wp}\,[\![C_2]\!]\,(f)$ after executing C_1. This gives the weakest preexpectation of $C_1;\, C_2$ with respect to postexpectation f.

The definition for the while loop makes use of a least fixed point, which is a standard construction in program semantics. Intuitively, the fixed point iteration of the wp–characteristic functional, given by 0, $F_f(0)$, $F_f^2(0)$, $F_f^3(0)$, $\ldots$, corresponds to the portion the expected value of f after termination of the loop, that can be collected within at most $0, 1, 2, 3, \ldots$ loop guard evaluations.

The Kleene Fixed Point Theorem [34] ensures that this iteration converges to the least fixed point, i.e.

$$\sup_{n\in\mathbb{N}} F_f^n(0) \;=\; \mathsf{lfp}\ F_f \;=\; \mathsf{wp}\,[\![\mathtt{while}\,(\varphi)\,\{C\}]\!]\,(f)\,.$$

By inspection of the above equality, we see that the least fixed point is exactly the construct that we want for while loops, since $\sup_{n\in\mathbb{N}} F_f^n(0)$ in principle allows the loop to run for any number of iterations, which captures precisely the semantics of a while loop, where the number of loop iterations is—in contrast to e.g. `for` loops—not determined upfront.

Finally, since $\mathtt{repeat}\,\{C\}\,\mathtt{until}\,(\varphi)$ is syntactic sugar for $C;\ \mathtt{while}\,(\varphi)\,\{C\}$, we simply define the weakest preexpectation of the former as the weakest preexpectation of the latter. Let us conclude our study of the effects of the wp transformer by means of an example:

Example 2. Consider the following program C:

$$c :\approx {}^1\!/_3 \cdot \langle 0\rangle + {}^2\!/_3 \cdot \langle 1\rangle;$$
$$\mathtt{if}\,(c = 0)\,\{x :\approx {}^1\!/_2 \cdot \langle 5\rangle + {}^1\!/_6 \cdot \langle y+1\rangle + {}^1\!/_3 \cdot \langle y-1\rangle\}\,\mathtt{else}\,\{\mathtt{skip}\}$$

Say we wish to reason about the expected value of $x + c$ after execution of the above program. We can do so by calculating $\mathsf{wp}\,[\![C]\!]\,(x+c)$ using the rules in Table 1. This calculation in the end yields $\mathsf{wp}\,[\![C]\!]\,(x+c) \;=\; {}^{3y+26}\!/_{18}$ The expected valuation of the expression $x + c$ after executing C is thus ${}^{3y+26}\!/_{18}$. Note that $x + c$ can be thought of as an expression that is evaluated in the final states after execution, whereas ${}^{3y+26}\!/_{18}$ must be evaluated in the initial state before execution of C. △

Healthiness Conditions of wp. The wp transformer enjoys some useful properties, sometimes called *healthiness conditions* [35]. Two of these healthiness conditions that we will heavily make use of are given below:

Theorem 1 (Healthiness Conditions for the wp Transformer [35]). *For all $C \in \mathsf{pGCL}$, $f_1, f_2 \in \mathbb{E}$, and $a \in \mathbb{R}_{\geq 0}$, the following holds:*

1. $\mathsf{wp}\,[\![C]\!]\,(a \cdot f_1 + f_2) \;=\; a \cdot \mathsf{wp}\,[\![C]\!]\,(f_1) + \mathsf{wp}\,[\![C]\!]\,(f_2)$ *(linearity)*
2. $\mathsf{wp}\,[\![C]\!]\,(0) \;=\; 0$ *(strictness).*

3.3 The Expected Runtime Transformer

While for deterministic programs we can speak of *the* runtime of a program on a given input, the situation is different for probabilistic programs: For those we instead have to speak of the *expected runtime* (ERT). Notice that the ERT can be finite (even constant) while the program may still admit infinite executions. An example of this is the geometric loop in Example 1.

A wp–like transformer designed specifically for reasoning about ERTs is the ert transformer [30]. Like wp, it is of type $\mathsf{ert}[\![C]\!] \colon \mathbb{E} \to \mathbb{E}$ and it can be shown that

Table 2. Rules for the ert–transformer.

C	$\mathsf{ert}\,[\![C]\!]\,(f)$
`skip`	$1 + f$
`diverge`	∞
$x :\approx \mu$	$1 + \lambda\sigma\bullet\ \int_{\mathbb{Q}} \big(\lambda v\bullet\ f[x/v]\big)\, d\mu_\sigma$
$\texttt{if}\,(\varphi)\,\{C_1\}\,\texttt{else}\,\{C_2\}$	$1 + [\varphi]\cdot \mathsf{ert}\,[\![C_1]\!]\,(f) + [\neg\varphi]\cdot \mathsf{ert}\,[\![C_2]\!]\,(f)$
$C_1;\, C_2$	$\mathsf{ert}\,[\![C_1]\!]\,\big(\big(\mathsf{ert}\,[\![C_2]\!]\,(f)\,\big)\big)$
$\texttt{while}\,(\varphi)\,\{C'\}$	$\mathsf{lfp}\ X\bullet\ 1 + [\neg\varphi]\cdot f + [\varphi]\cdot \mathsf{ert}\,[\![C']\!]\,(X)$
$\texttt{repeat}\,\{C'\}\,\texttt{until}\,(\varphi)$	$\mathsf{ert}\,[\![C';\,\texttt{while}\,(\neg\varphi)\,\{C'\}]\!]\,(f)$

$\mathsf{ert}\,[\![C]\!]\,(0)\,(\sigma)$ is precisely the *expected runtime of executing* C *on input* σ. More generally, if $f\colon \Sigma \to \mathbb{R}^{\infty}_{\geq 0}$ measures the time that is needed after executing C (thus f is evaluated in the final states after termination of C), then $\mathsf{ert}\,[\![C]\!]\,(f)\,(\sigma)$ is the expected time that is needed to run C on input σ and then let time f pass. For a more in–depth treatment of the ert transformer, see [30, Sect. 3]. The transformer is defined as follows:

Definition 3 (The ert Transformer [30]**).** *The expected runtime transformer* $\mathsf{ert}\colon \mathsf{pGCL} \to \mathbb{E} \to \mathbb{E}$ *is defined by induction on all* pGCL *programs according to the rules given in Table 2. We call* $F_f(X) = 1 + [\neg\varphi]\cdot f + [\varphi]\cdot \mathsf{wp}\,[\![C]\!]\,(X)$ *the* ert–characteristic functional *of the loop* $\texttt{while}\,(\varphi)\,\{C\}$ *with respect to postexpectation* f*. As with* wp*, for a given* ert*–characteristic function* F_f*, we call the sequence* $\{F_f^n(0)\}_{n\in\mathbb{N}}$ *the* orbit of F_f*. Notice that*

$$\mathsf{ert}\,[\![\texttt{while}\,(\varphi)\,\{C\}]\!]\,(f) \;=\; \mathsf{lfp}\ F_f \;=\; \sup\ \{F_f^n(0)\}_{n\in\mathbb{N}}.$$

The rules for ert are very similar to the rules for wp. The runtime model we assume is that `skip` statements, random assignments, and guard evaluations for both conditional choice and while loops cost one unit of time. This runtime model can easily be adopted to count only the number of loop iterations or only the number of random assignments, etc. We conclude with a strong connection between the wp and the ert transformer, that is crucial in our proofs:

Theorem 2 (Decomposition of ert [41]**).** *For any* $C \in \mathsf{pGCL}$ *and* $f \in \mathbb{E}$*,*

$$\mathsf{ert}\,[\![C]\!]\,(f) \;=\; \mathsf{ert}\,[\![C]\!]\,(0) + \mathsf{wp}\,[\![C]\!]\,(f)\,.$$

4 Expected Runtimes of i.i.d Loops

We derive a proof rule that allows to determine *exact ERTs of independent and identically distributed loops* (or *i.i.d. loops* for short). Intuitively, a loop

```
while ((x − 5)^2 + (y − 5)^2 ≥ 25) {
    x :≈ Unif[0 ... 10];
    y :≈ Unif[0 ... 10]
}
```

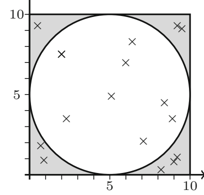


Fig. 2. An i.i.d. loop sampling a point within a circle uniformly at random using rejection sampling. The picture on the right–hand side visualizes the procedure: In each iteration a point ($\times$) is sampled. If we obtain a point within the white area inside the square, we terminate. Otherwise, i.e. if we obtain a point within the gray area outside the circle, we resample.

is i.i.d. if the distributions of states that are reached at the end of different loop iterations are equal. This is the case whenever there is no data flow across different iterations. In the non–probabilistic case, such loops either terminate after exactly one iteration or never. This is different for probabilistic programs.

As a running example, consider the program C_{circle} in Fig. 2. C_{circle} samples a point within a circle with center $(5, 5)$ and radius $r = 5$ uniformly at random using rejection sampling. In each iteration, it samples a point $(x, y) \in [0, \ldots, 10]^2$ within the square (with some fixed precision). The loop ensures that we resample if a sample is not located within the circle. Our proof rule will allow us to systematically determine the ERT of this loop, i.e. the average amount of time required until a single point within the circle is sampled.

Towards obtaining such a proof rule, we first present a syntactical notion of the i.i.d. property. It relies on expectations that are not affected by a pGCL program:

Definition 4. *Let $C \in$ pGCL and $f \in \mathbb{E}$. Moreover, let $\mathsf{Mod}(C)$ denote the set of all variables that occur on the left–hand side of an assignment in C, and let $\mathsf{Vars}(f)$ be the set of all variables that "occur in f", i.e. formally*

$$x \in \mathsf{Vars}(f) \qquad \text{iff} \qquad \exists\, \sigma\ \exists\, v, v' \colon \quad f(\sigma[x \mapsto v]) \;\neq\; f(\sigma[x \mapsto v']).$$

Then f is unaffected *by C, denoted $f \not\pitchfork C$, iff $\mathsf{Vars}(f) \cap \mathsf{Mod}(C) = \emptyset$.*

We are interested in expectations that are unaffected by pGCL programs because of a simple, yet useful observation: If $g \not\pitchfork C$, then g *can be treated like a constant* w.r.t. the transformer wp (i.e. like the a in Theorem 1 (1)). For our running example C_{circle} (see Fig. 2), the expectation $f = \mathsf{wp}\,[\![C_{body}]\!]\,([x + y \leq 10])$ is unaffected by the loop body C_{body} of C_{circle}. Consequently, we have $\mathsf{wp}\,[\![C_{body}]\!]\,(f) = f \cdot \mathsf{wp}\,[\![C_{body}]\!]\,(1) = f$. In general, we obtain the following property:

Lemma 1 (Scaling by Unaffected Expectations). *Let $C \in$ pGCL and $f, g \in \mathbb{E}$. Then $g \not\pitchfork C$ implies $\mathsf{wp}\,[\![C]\!]\,(g \cdot f) = g \cdot \mathsf{wp}\,[\![C]\!]\,(f)$.*

Proof. By induction on the structure of C. See [3, Appendix A.2]. □

We develop a proof rule that only requires that both the probability of the guard evaluating to true after one iteration of the loop body (i.e. $\mathsf{wp}\,[\![C]\!]\,([\varphi])$) as well as the expected value of $[\neg\varphi]\cdot f$ after one iteration (i.e. $\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)$) are unaffected by the loop body. We thus define the following:

Definition 5 (f–Independent and Identically Distributed Loops). *Let $C \in \mathsf{pGCL}$, φ be a guard, and $f \in \mathbb{E}$. Then we call the loop* $\mathtt{while}\,(\varphi)\,\{C\}$ f–independent and identically distributed *(or f–*i.i.d. *for short), if both*

$$\mathsf{wp}\,[\![C]\!]\,([\varphi]) \not\Vdash C \qquad \textit{and} \qquad \mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f) \not\Vdash C.$$

Example 3. Our example program C_{circle} (see Fig. 2) is f–i.i.d. for all $f \in \mathbb{E}$. This is due to the fact that

$$\mathsf{wp}\,[\![C_{body}]\!]\,\left(\left[(x-5)^2+(y-5)^2 \geq 25\right]\right) \;=\; \frac{48}{121} \not\Vdash C_{body} \qquad \text{(by Table 1)}$$

and (again for some fixed precision $p \in \mathbb{N}\setminus\{0\}$)

$$\begin{aligned}&\mathsf{wp}\,[\![C_{body}]\!]\,\left(\left[(x-5)^2+(y-5)^2 > 25\right]\cdot f\right)\\ &= \frac{1}{121}\cdot\sum_{i=0}^{10p}\sum_{j=0}^{10p}\left[(i/p-5)^2+(j/p-5)^2 > 25\right]\cdot f[x/(i/p),y/(j/p)] \not\Vdash C_{body}.\end{aligned} \quad \triangle$$

Our main technical Lemma is that we can express the orbit of the wp–characteristic function as a partial geometric series:

Lemma 2 (Orbits of f–i.i.d. Loops). *Let $C \in \mathsf{pGCL}$, φ be a guard, $f \in \mathbb{E}$ such that the loop* $\mathtt{while}\,(\varphi)\,\{C\}$ *is f–i.i.d, and let F_f be the corresponding* wp–*characteristic function. Then for all $n \in \mathbb{N}\setminus\{0\}$, it holds that*

$$F_f^n(0) \;=\; [\varphi]\cdot\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\cdot\sum_{i=0}^{n-2}\left(\mathsf{wp}\,[\![C]\!]\,([\varphi])^i\right) \;+\; [\neg\varphi]\cdot f.$$

Proof. By use of Lemma 1, see [3, Appendix A.3].

Using this precise description of the wp orbits, we now establish proof rules for f–i.i.d. loops, first for wp and later for ert.

Theorem 3 (Weakest Preexpectations of f–i.i.d. Loops). *Let $C \in \mathsf{pGCL}$, φ be a guard, and $f \in \mathbb{E}$. If the loop* $\mathtt{while}\,(\varphi)\,\{C\}$ *is f–i.i.d., then*

$$\mathsf{wp}\,[\![\mathtt{while}\,(\varphi)\,\{C\}]\!]\,(f) \;=\; [\varphi]\cdot\frac{\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)}{1-\mathsf{wp}\,[\![C]\!]\,([\varphi])} + [\neg\varphi]\cdot f\;,$$

where we define $\frac{0}{0} := 0$.

Proof. We have

$$
\begin{aligned}
&\mathsf{wp}\,[\![\mathtt{while}\,(\varphi)\,\{C\}]\!]\,(f) \\
&= \sup_{n\in\mathbb{N}}\ F_f^n(0) && \text{(by Definition 2)} \\
&= \sup_{n\in\mathbb{N}}\ [\varphi]\cdot\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\cdot\sum_{i=0}^{n-2}\left(\mathsf{wp}\,[\![C]\!]\,([\varphi])^i\right)+[\neg\varphi]\cdot f && \text{(by Lemma 2)} \\
&= [\varphi]\cdot\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\cdot\sum_{i=0}^{\omega}\left(\mathsf{wp}\,[\![C]\!]\,([\varphi])^i\right)+[\neg\varphi]\cdot f. && (\dagger)
\end{aligned}
$$

The preexpectation ($\dagger$) is to be evaluated in some state σ for which we have two cases: The first case is when $\mathsf{wp}\,[\![C]\!]\,([\varphi])\,(\sigma) < 1$. Using the closed form of the geometric series, i.e. $\sum_{i=0}^{\omega} q = \frac{1}{1-q}$ if $|q| < 1$, we get

$$
\begin{aligned}
&[\varphi]\,(\sigma)\cdot\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\,(\sigma)\cdot\sum_{i=0}^{\omega}\left(\mathsf{wp}\,[\![C]\!]\,([\varphi])\,(\sigma)^i\right)+[\neg\varphi]\,(\sigma)\cdot f(\sigma) \\
&&& \text{($\dagger$ instantiated in σ)} \\
&= [\varphi]\,(\sigma)\cdot\frac{\mathsf{wp}\,[\![C]\!]\,([\neg\varphi]\cdot f)\,(\sigma)}{1-\mathsf{wp}\,[\![C]\!]\,([\varphi])\,(\sigma)}+[\neg\varphi]\,(\sigma)\cdot f(\sigma). \\
&&& \text{(closed form of geometric series)}
\end{aligned}
$$

The second case is when $\mathsf{wp}\,[\![C]\!]\,([\varphi])\,(\sigma) = 1$. This case is technically slightly more involved. The full proof can be found in [3, Appendix A.4]. □

We now derive a similar proof rule for the ERT of an f–i.i.d. loop $\mathtt{while}\,(\varphi)\,\{C\}$.

Theorem 4 (Proof Rule for ERTs of f–i.i.d. Loops). *Let $C \in \mathsf{pGCL}$, φ be a guard, and $f \in \mathbb{E}$ such that all of the following conditions hold:*

1. $\mathtt{while}\,(\varphi)\,\{C\}$ *is f–i.i.d.*
2. $\mathsf{wp}\,[\![C]\!]\,(1) = 1$ *(loop body terminates almost–surely).*
3. $\mathsf{ert}\,[\![C]\!]\,(0) \not\Cap C$ *(every iteration runs in the same expected time).*

Then for the ERT of the loop $\mathtt{while}\,(\varphi)\,\{C\}$ w.r.t. postruntime f it holds that

$$\mathsf{ert}\,[\![\mathtt{while}\,(\varphi)\,\{C\}]\!]\,(f) \;=\; 1+\frac{[\varphi]\cdot(1+\mathsf{ert}\,[\![C]\!]\,([\neg\varphi]\cdot f))}{1-\mathsf{wp}\,[\![C]\!]\,([\varphi])}+[\neg\varphi]\cdot f\ ,$$

where we define $\frac{0}{0} := 0$ and $\frac{a}{0} := \infty$, for $a \neq 0$.

Proof. We first prove

$$\mathsf{ert}\,[\![\mathtt{while}\,(\varphi)\,\{C\}]\!]\,(0) \;=\; 1+[\varphi]\cdot\frac{1+\mathsf{ert}\,[\![C]\!]\,(0)}{1-\mathsf{wp}\,[\![C]\!]\,([\varphi])}. \qquad (\ddagger)$$

To this end, we propose the following expression as the orbit of the ert–characteristic function of the loop w.r.t. 0:

$$F_0^n(0) \;=\; 1 + [\varphi] \cdot \left(\mathsf{ert}\,[\![C]\!]\,(0) \cdot \sum_{i=0}^{n} \mathsf{wp}\,[\![C]\!]\,([\varphi])^i \;+\; \sum_{i=0}^{n-1} \mathsf{wp}\,[\![C]\!]\,([\varphi])^i \right)$$

For a verification that the above expression is indeed the correct orbit, we refer to the rigorous proof of this theorem in [3, Appendix A.5]. Now, analogously to the reasoning in the proof of Theorem 3 (i.e. using the closed form of the geometric series and case distinction on whether $\mathsf{wp}\,[\![C]\!]\,([\varphi]) < 1$ or $\mathsf{wp}\,[\![C]\!]\,([\varphi]) = 1$), we get that the supremum of this orbit is indeed the right–hand side of (‡). To complete the proof, consider the following:

$$\begin{aligned}
&\mathsf{ert}\,[\![\mathtt{while}\,(\varphi)\,\{C\}]\!]\,(f) \\
&= \mathsf{ert}\,[\![\mathtt{while}\,(\varphi)\,\{C\}]\!]\,(0) + \mathsf{wp}\,[\![\mathtt{while}\,(\varphi)\,\{C\}]\!]\,(f) && \text{(by Theorem 2)} \\
&= 1 + [\varphi] \cdot \frac{1 + \mathsf{ert}\,[\![C]\!]\,(0)}{1 - \mathsf{wp}\,[\![C]\!]\,([\varphi])} + [\varphi] \cdot \frac{\mathsf{wp}\,[\![C]\!]\,([\neg\varphi] \cdot f)}{1 - \mathsf{wp}\,[\![C]\!]\,([\varphi])} + [\neg\varphi] \cdot f \\
& && \text{(by (‡) and Theorem 3)} \\
&= 1 + [\varphi] \cdot \frac{1 + \mathsf{ert}\,[\![C]\!]\,([\neg\varphi] \cdot f)}{1 - \mathsf{wp}\,[\![C]\!]\,([\varphi])} + [\neg\varphi] \cdot f && \text{(by Theorem 2)}
\end{aligned}$$

□

5 A Programming Language for Bayesian Networks

So far we have derived proof rules for formal reasoning about expected outcomes and expected run-times of i.i.d. loops (Theorems 3 and 4). In this section, we apply these results to develop a syntactic pGCL fragment that allows exact computations of closed forms of ERTs. In particular, no invariants, (super)martingales or fixed point computations are required.

After that, we show how BNs with observations can be translated into pGCL programs within this fragment. Consequently, we call our pGCL fragment the *Bayesian Network Language*. As a result of the above translation, we obtain a systematic and automatable approach to compute the *expected sampling time* of a BN in the presence of observations. That is, the expected time it takes to obtain a single sample that satisfies all observations.

5.1 The Bayesian Network Language

Programs in the Bayesian Network Language are organized as sequences of blocks. Every block is associated with a single variable, say x, and satisfies two constraints: First, no variable other than x is modified inside the block, i.e. occurs on the left–hand side of a random assignment. Second, every variable accessed inside of a guard has been initialized before. These restrictions ensure that there is no data flow across multiple executions of the same block. Thus, intuitively, all loops whose body is composed from blocks (as described above) are f–i.i.d. loops.

Definition 6 (The Bayesian Network Language). *Let* $\mathsf{Vars} = \{x_1, x_2, \ldots\}$ *be a finite set of program variables as in Sect. 3. The set of programs in Bayesian Network Language, denoted* BNL*, is given by the grammar*

$$\begin{aligned} C &\longrightarrow \mathit{Seq} \quad | \quad \texttt{repeat}\,\{\mathit{Seq}\}\,\texttt{until}\,(\psi) \quad | \quad C;\, C \\ \mathit{Seq} &\longrightarrow \mathit{Seq};\, \mathit{Seq} \quad | \quad B_{x_1} \quad | \quad B_{x_2} \quad | \quad \ldots \\ B_{x_i} &\longrightarrow x_i :\approx \mu \quad | \quad \texttt{if}\,(\varphi)\,\{x_i :\approx \mu\}\,\texttt{else}\,\{B_{x_i}\} \end{aligned}$$

(rule exists for all $x_i \in \mathsf{Vars}$)

where $x_i \in \mathsf{Vars}$ *is a program variable, all variables in* φ *have been initialized before, and* B_{x_i} *is a non–terminal parameterized with program variable* $x_i \in \mathsf{Vars}$*. That is, for all* $x_i \in \mathsf{Vars}$ *there is a non–terminal* B_{x_i}*. Moreover,* ψ *is an arbitrary guard and* μ *is a distribution expression of the form* $\mu = \sum_{j=1}^{n} p_j \cdot \langle a_j \rangle$ *with* $a_j \in \mathbb{Q}$ *for* $1 \leq j \leq n$*.*

Example 4. Consider the BNL program C_{dice}:

$$x_1 :\approx \texttt{Unif}[1 \ldots 6];\; \texttt{repeat}\,\{x_2 :\approx \texttt{Unif}[1 \ldots 6]\}\,\texttt{until}\,(x_2 \geq x_1)$$

This program first throws a fair die. After that it keeps throwing a second die until its result is at least as large as the first die. △

For any $C \in \mathsf{BNL}$, our goal is to compute the exact ERT of C, i.e. $\mathsf{ert}\,[\![C]\!]\,(0)$. In case of loop–free programs, this amounts to a straightforward application of the ert calculus presented in Sect. 3. To deal with loops, however, we have to perform fixed point computations or require user–supplied artifacts, e.g. invariants, supermartingales, etc. For BNL programs, on the other hand, it suffices to apply the proof rules developed in Sect. 4. As a result, we directly obtain an exact closed form solution for the ERT of a loop. This is a consequence of the fact that all loops in BNL are f–i.i.d., which we establish in the following.

By definition, every loop in BNL is of the form $\texttt{repeat}\,\{B_{x_i}\}\,\texttt{until}\,(\psi)$, which is equivalent to $B_{x_i};\, \texttt{while}\,(\neg\psi)\,\{B_{x_i}\}$. Hence, we want to apply Theorem 4 to that while loop. Our first step is to discharge the theorem's premises:

Lemma 3. *Let Seq be a sequence of* BNL*–blocks,* $g \in \mathbb{E}$*, and* ψ *be a guard. Then:*

1. *The expected value of g after executing Seq is unaffected by Seq. That is,* $\mathsf{wp}\,[\![\mathit{Seq}]\!]\,(g) \not\Cap \mathit{Seq}$*.*
2. *The ERT of Seq is unaffected by Seq, i.e.* $\mathsf{ert}\,[\![\mathit{Seq}]\!]\,(0) \not\Cap \mathit{Seq}$*.*
3. *For every* $f \in \mathbb{E}$*, the loop* $\texttt{while}\,(\neg\psi)\,\{\mathit{Seq}\}$ *is* f*–i.i.d.*

Proof. 1. is proven by induction on the length of the sequence of blocks *Seq* and 2. is a consequence of 1., see [3, Appendix A.6]. 3. follows immediately from 1. by instantiating g with $[\neg\psi]$ and $[\psi] \cdot f$, respectively. □

We are now in a position to derive a closed form for the ERT of loops in BNL.

Theorem 5. *For every loop* $\texttt{repeat}\,\{Seq\}\,\texttt{until}\,(\psi) \in \mathsf{BNL}$ *and every* $f \in \mathbb{E}$,

$$\mathsf{ert}\,[\![\texttt{repeat}\,\{Seq\}\,\texttt{until}\,(\psi)]\!]\,(f) \;=\; \frac{1 + \mathsf{ert}\,[\![Seq]\!]\,([\psi]\cdot f)}{\mathsf{wp}\,[\![Seq]\!]\,([\psi])}.$$

Proof. Let $f \in \mathbb{E}$. Moreover, recall that $\texttt{repeat}\,\{Seq\}\,\texttt{until}\,(\psi)$ is equivalent to the program $Seq;\ \texttt{while}\,(\neg\psi)\,\{Seq\} \in \mathsf{BNL}$. Applying the semantics of ert (Table 2), we proceed as follows:

$$\mathsf{ert}\,[\![\texttt{repeat}\,\{Seq\}\,\texttt{until}\,(\psi)]\!]\,(f) \;=\; \mathsf{ert}\,[\![Seq]\!]\,(\mathsf{ert}\,[\![\texttt{while}\,(\neg\psi)\,\{Seq\}]\!]\,(f))$$

Since the loop body Seq is loop–free, it terminates certainly, i.e. $\mathsf{wp}\,[\![Seq]\!]\,(1) = 1$ (Premise 2. of Theorem 4). Together with Lemma 3.1. and 3., all premises of Theorem 4 are satisfied. Hence, we obtain a closed form for $\mathsf{ert}\,[\![\texttt{while}\,(\neg\psi)\,\{Seq\}]\!]\,(f)$:

$$= \;\mathsf{ert}[\![Seq]\!]\left(\underbrace{1 + \frac{[\neg\psi]\cdot(1 + \mathsf{ert}\,[\![Seq]\!]\,([\psi]\cdot f))}{1 - \mathsf{wp}\,[\![Seq]\!]\,([\neg\psi])} + [\psi]\cdot f}_{=:g}\right)$$

By Theorem 2, we know $\mathsf{ert}\,[\![Seq]\!]\,(g) = \mathsf{ert}\,[\![Seq]\!]\,(0) + \mathsf{wp}\,[\![C]\!]\,(g)$ for any g. Thus:

$$= \;\mathsf{ert}\,[\![Seq]\!]\,(0) + \mathsf{wp}[\![Seq]\!]\left(\underbrace{1 + \frac{[\neg\psi]\cdot(1 + \mathsf{ert}\,[\![Seq]\!]\,([\psi]\cdot f))}{1 - \mathsf{wp}\,[\![Seq]\!]\,([\neg\psi])} + [\psi]\cdot f}_{g}\right)$$

Since wp is linear (Theorem 1 (2)), we obtain:

$$\begin{aligned} = \;& \mathsf{ert}\,[\![Seq]\!]\,(0) + \underbrace{\mathsf{wp}\,[\![Seq]\!]\,(1)}_{=\,1} + \mathsf{wp}\,[\![Seq]\!]\,([\psi]\cdot f) \\ & + \mathsf{wp}\,[\![Seq]\!]\left(\frac{[\neg\psi]\cdot(1 + \mathsf{ert}\,[\![Seq]\!]\,([\psi]\cdot f))}{1 - \mathsf{wp}\,[\![Seq]\!]\,([\neg\psi])}\right) \end{aligned}$$

By a few simple algebraic transformations, this coincides with:

$$= \;1 + \mathsf{ert}\,[\![Seq]\!]\,(0) + \mathsf{wp}\,[\![Seq]\!]\,([\psi]\cdot f) + \mathsf{wp}\,[\![Seq]\!]\left([\neg\psi]\cdot\frac{1 + \mathsf{ert}\,[\![Seq]\!]\,([\psi]\cdot f)}{1 - \mathsf{wp}\,[\![Seq]\!]\,([\neg\psi])}\right)$$

Let R denote the fraction above. Then Lemma 3.1. and 2. implies $R \not\!\!\!\not\curvearrowright Seq$. We may thus apply Lemma 1 to derive $\mathsf{wp}\,[\![Seq]\!]\,([\neg\psi]\cdot R) = \mathsf{wp}\,[\![Seq]\!]\,([\neg\psi])\cdot R$. Hence:

$$= \;1 + \mathsf{ert}\,[\![Seq]\!]\,(0) + \mathsf{wp}\,[\![Seq]\!]\,([\psi]\cdot f) + \mathsf{wp}\,[\![Seq]\!]\,([\neg\psi])\cdot\frac{1 + \mathsf{ert}\,[\![Seq]\!]\,([\psi]\cdot f)}{1 - \mathsf{wp}\,[\![Seq]\!]\,([\neg\psi])}$$

Again, by Theorem 2, we know that $\mathsf{ert}\,[\![Seq]\!]\,(g) = \mathsf{ert}\,[\![Seq]\!]\,(0) + \mathsf{wp}\,[\![Seq]\!]\,(g)$ for any g. Thus, for $g = [\psi]\cdot f$, this yields:

$$= \;1 + \mathsf{ert}\,[\![Seq]\!]\,([\psi]\cdot f) + \mathsf{wp}\,[\![Seq]\!]\,([\neg\psi])\cdot\frac{1 + \mathsf{ert}\,[\![Seq]\!]\,([\psi]\cdot f)}{1 - \mathsf{wp}\,[\![Seq]\!]\,([\neg\psi])}$$

Then a few algebraic transformations lead us to the claimed ERT:

$$= \frac{1 + \mathsf{ert}\,[\![\mathit{Seq}]\!]\,([\psi] \cdot f)}{\mathsf{wp}\,[\![\mathit{Seq}]\!]\,([\psi])}. \qquad \square$$

Note that Theorem 5 holds for arbitrary postexpectations $f \in \mathbb{E}$. This enables *compositional reasoning* about ERTs of BNL programs. Since all other rules of the ert–calculus for loop–free programs amount to simple syntactical transformations (see Table 2), we conclude that

Corollary 1. *For any $C \in \mathsf{BNL}$, a closed form for $\mathsf{ert}\,[\![C]\!]\,(0)$ can be computed compositionally.*

Example 5. Theorem 5 allows us to comfortably compute the ERT of the BNL program C_{dice} introduced in Example 4:

$$x_1 :\approx \texttt{Unif}[1 \ldots 6];\ \texttt{repeat}\,\{x_2 :\approx \texttt{Unif}[1 \ldots 6]\}\,\texttt{until}\,(x_2 \geq x_1)$$

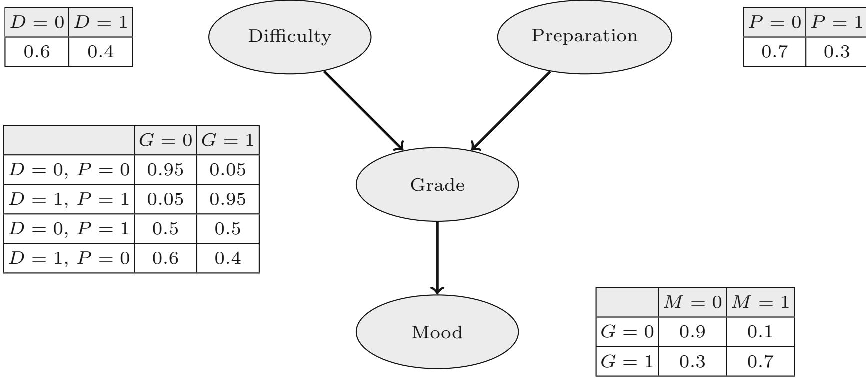
For the ERT, we have

$$\begin{aligned}
& \mathsf{ert}\,[\![C_{dice}]\!]\,(0) \\
=\ & \mathsf{ert}\,[\![x_1 :\approx \texttt{Unif}[1 \ldots 6]]\!]\,(\mathsf{ert}\,[\![\texttt{repeat}\,\{\ldots\}\,\texttt{until}\,([x_2 \geq x_1])]\!]\,(0)) && \text{(Table 2)} \\
=\ & \mathsf{ert}\,[\![x_1 :\approx \texttt{Unif}[1 \ldots 6]]\!] \left(\frac{1 + \mathsf{ert}\,[\![x_2 :\approx \texttt{Unif}[1 \ldots 6]]\!]\,([x_2 \geq x_1])}{\mathsf{wp}\,[\![x_1 :\approx \texttt{Unif}[1 \ldots 6]]\!]\,([x_2 \geq x_1])} \right) && \text{(Thm. 5)} \\
=\ & \sum_{1 \leq i \leq 6} {}^{1}\!/_{6} \cdot \frac{1 + \sum_{1 \leq j \leq 6} {}^{1}\!/_{6} \cdot [j \geq i]}{\sum_{1 \leq j \leq 6} {}^{1}\!/_{6} \cdot [j \geq i]} && \text{(Table 2)} \\
=\ & 3.45. && \triangle
\end{aligned}$$

5.2 Bayesian Networks

To reason about expected sampling times of BNs, it remains to develop a sound translation from BNs with observations into equivalent BNL programs. A BN is a probabilistic graphical model that is given by a directed acyclic graph. Every node is a random variable and a directed edge between two nodes expresses a probabilistic dependency between these nodes.

As a running example, consider the BN depicted in Fig. 3 (inspired by [31]) that models the mood of students after taking an exam. The network contains four random variables. They represent the difficulty of the exam (D), the level of preparation of a student (P), the achieved grade (G), and the resulting mood (M). For simplicity, let us assume that each random variable assumes either 0 or 1. The edges express that the student's mood depends on the achieved grade which, in turn, depends on the difficulty of the exam and the preparation of the student. Every node is accompanied by a table that provides the conditional probabilities of a node *given* the values of all the nodes it depends upon. We can then use the BN to answer queries such as "What is the probability that a



$D=0$	$D=1$
0.6	0.4

$P=0$	$P=1$
0.7	0.3

	$G=0$	$G=1$
$D=0,\ P=0$	0.95	0.05
$D=1,\ P=1$	0.05	0.95
$D=0,\ P=1$	0.5	0.5
$D=1,\ P=0$	0.6	0.4

	$M=0$	$M=1$
$G=0$	0.9	0.1
$G=1$	0.3	0.7

Fig. 3. A Bayesian network

student is well–prepared for an exam ($P = 1$), but ends up with a bad mood ($M = 0$)?"

In order to translate BNs into equivalent BNL programs, we need a formal representation first. Technically, we consider *extended* BNs in which nodes may additionally depend on inputs that are not represented by nodes in the network. This allows us to define a compositional translation without modifying conditional probability tables.

Towards a formal definition of extended BNs, we use the following notation. A tuple $(s_1, \ldots, s_k) \in S^k$ of length k over some set S is denoted by $\mathbf{s}$. The empty tuple is ε. Moreover, for $1 \leq i \leq k$, the i-th element of tuple $\mathbf{s}$ is given by $\mathbf{s}(i)$. To simplify the presentation, we assume that all nodes and all inputs are represented by natural numbers.

Definition 7. *An* extended Bayesian network*, EBN for short, is a tuple* $\mathcal{B} = (V, I, E, \mathsf{Vals}, \mathsf{dep}, \mathsf{cpt})$*, where*

- $V \subseteq \mathbb{N}$ *and* $I \subseteq \mathbb{N}$ *are finite disjoint sets of* nodes *and* inputs.
- $E \subseteq V \times V$ *is a set of* edges *such that* (V, E) *is a directed acyclic graph.*
- Vals *is a finite set of possible* values *that can be assigned to each node.*
- $\mathsf{dep} \colon V \rightarrow (V \cup I)^*$ *is a function assigning each node* v *to an ordered sequence of* dependencies*. That is,* $\mathsf{dep}(v) = (u_1, \ldots, u_m)$ *such that* $u_i < u_{i+1}$ $(1 \leq i < m)$*. Moreover, every dependency* u_j $(1 \leq j \leq m)$ *is either an input, i.e.* $u_j \in I$*, or a node with an edge to* v*, i.e.* $u_j \in V$ *and* $(u_j, v) \in E$.
- cpt *is a function mapping each node* v *to its* conditional probability table $\mathsf{cpt}[v]$*. That is, for* $k = |\mathsf{dep}(v)|$*,* $\mathsf{cpt}[v]$ *is given by a function of the form*

$$\mathsf{cpt}[v] : \mathsf{Vals}^k \rightarrow \mathsf{Vals} \rightarrow [0,1] \quad \text{such that} \quad \sum_{\mathbf{z} \in \mathsf{Vals}^k, a \in \mathsf{Vals}} \mathsf{cpt}[v](\mathbf{z})(a) \; = \; 1.$$

Here, the i*-th entry in a tuple* $\mathbf{z} \in \mathsf{Vals}^k$ *corresponds to the value assigned to the* i*-th entry in the sequence of dependencies* $\mathsf{dep}(v)$.

A Bayesian network *(BN) is an extended BN without inputs, i.e.* $I = \emptyset$*. In particular, the dependency function is of the form* $\mathsf{dep}\colon V \to V^*$*.*

Example 6. The formalization of our example BN (Fig. 3) is straightforward. For instance, the dependencies of variable G are given by $\mathsf{dep}(G) = (D, P)$ (assuming D is encoded by an integer less than P). Furthermore, every entry in the conditional probability table of node G corresponds to an evaluation of the function $\mathsf{cpt}[G]$. For example, if $D = 1$, $P = 0$, and $G = 1$, we have $\mathsf{cpt}[G](1,0)(1) = 0.4$. △

In general, the conditional probability table cpt determines the conditional probability distribution of each node $v \in V$ given the nodes and inputs it depends on. Formally, we interpret an entry in a conditional probability table as follows:

$$\mathsf{Pr}\left(v = a \mid \mathsf{dep}(v) = \mathbf{z}\right) \;=\; \mathsf{cpt}[v](\mathbf{z})(a),$$

where $v \in V$ is a node, $a \in \mathsf{Vals}$ is a value, and $\mathbf{z}$ is a tuple of values of length $|\mathsf{dep}(v)|$. Then, by the chain rule, the joint probability of a BN is given by the product of its conditional probability tables (cf. [4]).

Definition 8. *Let* $\mathcal{B} = (V, I, E, \mathsf{Vals}, \mathsf{dep}, \mathsf{cpt})$ *be an extended Bayesian network. Moreover, let* $W \subseteq V$ *be a downward closed*[5] *set of nodes. With each* $w \in W \cup I$*, we associate a fixed value* $\underline{w} \in \mathsf{Vals}$*. This notation is lifted pointwise to tuples of nodes and inputs. Then the* joint probability *in which nodes in* W *assume values* $\underline{W}$ *is given by*

$$\mathsf{Pr}\left(W = \underline{W}\right) \;=\; \prod_{v \in W} \mathsf{Pr}\left(v = \underline{v} \mid \mathsf{dep}(v) = \underline{\mathsf{dep}(v)}\right) \;=\; \prod_{v \in W} \mathsf{cpt}[v](\underline{\mathsf{dep}(v)})(\underline{v}).$$

The conditional joint probability distribution of a set of nodes W*, given observations on a set of nodes* O*, is then given by the quotient* $\mathsf{Pr}(W = \underline{W}) / \mathsf{Pr}(O = \underline{O})$*.*

For example, the probability of a student having a bad mood, i.e. $M = 0$, after getting a bad grade ($G = 0$) for an easy exam ($D = 0$) given that she was well–prepared, i.e. $P = 1$, is

$$\begin{aligned} \mathsf{Pr}\left(D = 0, G = 0, M = 0 \mid P = 1\right) \;&=\; \frac{\mathsf{Pr}\left(D = 0, G = 0, M = 0, P = 1\right)}{\mathsf{Pr}\left(P = 1\right)} \\ &=\; \frac{0.9 \cdot 0.5 \cdot 0.6 \cdot 0.3}{0.3} \;=\; 0.27. \end{aligned}$$

5.3 From Bayesian Networks to BNL

We now develop a compositional translation from EBNs into BNL programs. Throughout this section, let $\mathcal{B} = (V, I, E, \mathsf{Vals}, \mathsf{dep}, \mathsf{cpt})$ be a fixed EBN. Moreover, with every node or input $v \in V \cup I$ we associate a program variable x_v.

We proceed in three steps: First, *every node together with its dependencies* is translated into a *block* of a BNL program. These blocks are then composed into a single BNL program that captures the whole BN. Finally, we implement conditioning by means of rejection sampling.

[5] W is downward closed if $v \in W$ and $(u, v) \in E$ implies $u \in E$.

Step 1: We first present the atomic building blocks of our translation. Let $v \in V$ be a node. Moreover, let $\mathbf{z} \in \mathsf{Vals}^{|\mathsf{dep}(v)|}$ be an evaluation of the dependencies of v. That is, $\mathbf{z}$ is a tuple that associates a value with every node and input that v depends on (in the same order as $\mathsf{dep}(v)$). For every node v and evaluation of its dependencies $\mathbf{z}$, we define a corresponding guard and a random assignment:

$$\begin{aligned} guard_{\mathcal{B}}(v, \mathbf{z}) &= \bigwedge_{1 \leq i \leq |\mathsf{dep}(v)|} x_{\mathsf{dep}(v)(i)} = \mathbf{z}(i) \\ assign_{\mathcal{B}}(v, \mathbf{z}) &= x_v :\approx \sum_{a \in \mathsf{Vals}} \mathsf{cpt}[v](\mathbf{z})(a) \cdot \langle a \rangle \end{aligned}$$

Note that $\mathsf{dep}(v)(i)$ is the i-th element from the sequence of nodes $\mathsf{dep}(v)$.

Example 7. Continuing our previous example (see Fig. 1), assume we fixed the node $v = G$. Moreover, let $\mathbf{z} = (1, 0)$ be an evaluation of $\mathsf{dep}(v) = (S, R)$. Then the guard and assignment corresponding to v and $\mathbf{z}$ are given by:

$$\begin{aligned} guard_{\mathcal{B}}(G, (1, 0)) &= x_D = 1 \ \wedge x_P = 0, \text{ and} \\ assign_{\mathcal{B}}(G, (1, 0)) &= x_G :\approx 0.6 \cdot \langle 0 \rangle + 0.4 \cdot \langle 1 \rangle. \end{aligned}$$

△

We then translate every node $v \in V$ into a program block that uses guards to determine the rows in the conditional probability table under consideration. After that, the program samples from the resulting probability distribution using the previously constructed assignments. In case a node does neither depend on other nodes nor input variables we omit the guards. Formally,

$$block_{\mathcal{B}}(v) = \begin{cases} assign_{\mathcal{B}}(v, \varepsilon) & \text{if } |\mathsf{dep}(v)| = 0 \\ \texttt{if}\,(guard_{\mathcal{B}}(v, \mathbf{z_1}))\,\{ & \\ \quad assign_{\mathcal{B}}(v, \mathbf{z_1})\} & \\ \texttt{else}\,\{\texttt{if}\,(guard_{\mathcal{B}}(v, \mathbf{z_2}))\,\{ & \text{if } |\mathsf{dep}(v)| = k > 0 \\ \quad assign_{\mathcal{B}}(v, \mathbf{z_2})\} & \text{and } \mathsf{Vals}^k = \{\mathbf{z_1}, \ldots, \mathbf{z_m}\}. \\ \ldots\}\ \texttt{else}\ \{ & \\ \quad assign_{\mathcal{B}}(v, \mathbf{z_m})\}\ldots\} & \end{cases}$$

Remark 1. The guards under consideration are conjunctions of equalities between variables and literals. We could thus use a more efficient translation of conditional probability tables by adding a `switch-case` statement to our probabilistic programming language. Such a statement is of the form

$$\texttt{switch}(\mathbf{x})\ \{\ \texttt{case}\ \mathbf{a}_1 : C_1\ \texttt{case}\ a_2 : C_2\ \ldots\ \texttt{default} : C_m\},$$

where $\mathbf{x}$ is a tuple of variables, and $\mathbf{a}_1, \ldots \mathbf{a}_{m-1}$ are tuples of rational numbers of the same length as $\mathbf{x}$. With respect to the `wp` semantics, a `switch-case` statement is syntactic sugar for nested `if-then-else` blocks as used in the above translation. However, the runtime model of a `switch-case` statement requires just a single guard evaluation (φ) instead of potentially multiple guard evaluations when evaluating nested `if-then-else` blocks. Since the above adaption is straightforward, we opted to use nested `if-then-else` blocks to keep our programming language simple and allow, in principle, more general guards. △

Step 2: The next step is to translate a complete EBN into a BNL program. To this end, we compose the blocks obtained from each node starting at the roots of the network. That is, all nodes that contain no incoming edges. Formally,

$$roots(\mathcal{B}) = \{v \in V_{\mathcal{B}} \mid \neg\exists u \in V_{\mathcal{B}}\colon (u,v) \in E_{\mathcal{B}}\}.$$

After translating every node in the network, we remove them from the graph, i.e. every root becomes an input, and proceed with the translation until all nodes have been removed. More precisely, given a set of nodes $S \subseteq V$, the extended BN $\mathcal{B} \setminus S$ obtained by removing S from $\mathcal{B}$ is defined as

$$\mathcal{B} \setminus S \;=\; (V \setminus S,\, I \cup S,\, E \setminus (V \times S \cup S \times V),\, \mathsf{dep},\, \mathsf{cpt})\,.$$

With these auxiliary definitions readily available, an extended BN $\mathcal{B}$ is translated into a BNL program as follows:

$$BNL(\mathcal{B}) \;=\; \begin{cases} block_{\mathcal{B}}(r_1); \ldots; block_{\mathcal{B}}(r_m) & \text{if } roots(\mathcal{B}) = \{r_1, \ldots, r_m\} = V \\ block_{\mathcal{B}}(r_1); \ldots; block_{\mathcal{B}}(r_m); & \text{if } roots(\mathcal{B}) = \{r_1, \ldots, r_m\} \subsetneq V \\ BNL(\mathcal{B} \setminus roots(\mathcal{B})) & \end{cases}$$

Step 3: To complete the translation, it remains to account for observations. Let $cond\colon V \to \mathsf{Vals} \cup \{\bot\}$ be a function mapping every node either to an observed value in Vals or to $\bot$. The former case is interpreted as an observation that node v has value $cond(v)$. Otherwise, i.e. if $cond(v) = \bot$, the value of node v is *not observed*. We collect all observed nodes in the set $O = \{v \in V \mid cond(v) \neq \bot\}$. It is then natural to incorporate conditioning into our translation by applying rejection sampling: We repeatedly execute a BNL program until every observed node has the desired value $cond(v)$. In the presence of observations, we translate the extended BN $\mathcal{B}$ into a BNL program as follows:

$$BNL(\mathcal{B}, cond) \;=\; \texttt{repeat}\,\{BNL(\mathcal{B})\}\,\texttt{until}\left(\bigwedge_{v \in O} x_v = cond(v)\right)$$

Example 8. Consider, again, the BN $\mathcal{B}$ depicted in Fig. 3. Moreover, assume we observe $P = 1$. Hence, the conditioning function $cond$ is given by $cond(P) = 1$ and $cond(v) = \bot$ for $v \in \{D, G, M\}$. Then the translation of $\mathcal{B}$ and $cond$, i.e. $BNL(\mathcal{B}, cond)$, is the BNL program C_{mood} depicted in Fig. 4. △

Since our translation yields a BNL program for any given BN, we can compositionally compute a closed form for the expected simulation time of a BN. This is an immediate consequence of Corollary 1.

We still have to prove, however, that our translation is sound, i.e. the conditional joint probabilities inferred from a BN coincide with the (conditional) joint probabilities from the corresponding BNL program. Formally, we obtain the following soundness result.

```
1  repeat {                                      10     } else {
2      x_D :≈ 0.6 · ⟨0⟩ + 0.4 · ⟨1⟩;              11         x_G :≈ 0.6 · ⟨0⟩ + 0.4 · ⟨1⟩
3      x_P :≈ 0.7 · ⟨0⟩ + 0.3 · ⟨1⟩               12     };
4      if (x_D = 0 ∧ x_P = 0) {                   13     if (x_G = 0) {
5          x_G :≈ 0.95 · ⟨0⟩ + 0.05 · ⟨1⟩         14         x_M :≈ 0.9 · ⟨0⟩ + 0.1 · ⟨1⟩
6      } else if (x_D = 1 ∧ x_P = 1) {            15     } else {
7          x_G :≈ 0.05 · ⟨0⟩ + 0.95 · ⟨1⟩         16         x_M :≈ 0.3 · ⟨0⟩ + 0.7 · ⟨1⟩
8      } else if (x_D = 0 ∧ x_P = 1) {            17     }
9          x_G :≈ 0.5 · ⟨0⟩ + 0.5 · ⟨1⟩           18  } until (x_P = 1)
```

Fig. 4. The BNL program C_{mood} obtained from the BN in Fig. 3.

Theorem 6 (Soundness of Translation). *Let $\mathcal{B} = (V, I, E, \mathsf{Vals}, \mathsf{dep}, \mathsf{cpt})$ be a BN and $cond : V \to \mathsf{Vals} \cup \{\bot\}$ be a function determining the observed nodes. For each node and input v, let $\underline{v} \in \mathsf{Vals}$ be a fixed value associated with v. In particular, we set $\underline{v} = cond(v)$ for each observed node $v \in O$. Then*

$$\mathsf{wp}\,[\![BNL(\mathcal{B}, cond)]\!]\left(\left[\bigwedge_{v \in V \setminus O} x_v = \underline{v}\right]\right) = \frac{\mathsf{Pr}\left(\bigwedge_{v \in V} v = \underline{v}\right)}{\mathsf{Pr}\left(\bigwedge_{o \in O} o = \underline{o}\right)}.$$

Proof. Without conditioning, i.e. $O = \emptyset$, the proof proceeds by induction on the number of nodes of $\mathcal{B}$. With conditioning, we additionally apply Theorems 3 and 5 to deal with loops introduced by observed nodes. See [3, Appendix A.7]. □

Example 9 (Expected Sampling Time of a BN). Consider, again, the BN $\mathcal{B}$ in Fig. 3. Moreover, recall the corresponding program C_{mood} derived from $\mathcal{B}$ in Fig. 4, where we observed $P = 1$. By Theorem 6 we can also determine the probability that a student who got a bad grade in an easy exam was well–prepared by means of weakest precondition reasoning. This yields

$$\begin{aligned} &\mathsf{wp}\,[\![C_{mood}]\!]\left([x_D = 0 \wedge x_G = 0 \wedge x_M = 0]\right) \\ &= \frac{\mathsf{Pr}\left(D = 0, G = 0, M = 0, P = 1\right)}{\mathsf{Pr}\left(P = 1\right)} = 0.27. \end{aligned}$$

Furthermore, by Corollary 1, it is straightforward to determine the expected time to obtain a single sample of $\mathcal{B}$ that satisfies the observation $P = 1$:

$$\mathsf{ert}\,[\![C_{mood}]\!]\,(0) = \frac{1 + \mathsf{ert}\,[\![C_{loop\text{-}body}]\!]\,(0)}{\mathsf{wp}\,[\![C_{loop\text{-}body}]\!]\,([P = 1])} = 23.4 + {}^{1}\!/_{15} = 23.4\overline{6}. \qquad \triangle$$

6 Implementation

We implemented a prototype in Java to analyze expected sampling times of Bayesian networks. More concretely, our tool takes as input a BN together with

observations in the popular Bayesian Network Interchange Format.[6] The BN is then translated into a BNL program as shown in Sect. 5. Our tool applies the ert–calculus together with our proof rules developed in Sect. 4 to compute the exact expected runtime of the BNL program.

The size of the resulting BNL program is linear in the total number of rows of all conditional probability tables in the BN. The program size is thus *not* the bottleneck of our analysis. As we are dealing with an NP–hard problem [12,13], it is not surprising that our algorithm has a worst–case exponential time complexity. However, also the space complexity of our algorithm is exponential in the worst case: As an expectation is propagated backwards through an `if`–clause of the BNL program, the size of the expectation is potentially multiplied. This is also the reason that our analysis runs out of memory on some benchmarks.

We evaluated our implementation on the *largest* BNs in the Bayesian Network Repository [46] that consists—to a large extent—of real–world BNs including expert systems for, e.g., electromyography (`munin`) [2], hematopathology diagnosis (`hepar2`) [42], weather forecasting (`hailfinder`) [1], and printer troubleshooting in Windows 95 (`win95pts`) [45, Sect. 5.6.2]. For a evaluation of *all* BNs in the repository, we refer to the extended version of this paper [3, Sect. 6].

All experiments were performed on an HP BL685C G7. Although up to 48 cores with 2.0 GHz were available, only one core was used apart from Java's garbage collection. The Java virtual machine was limited to 8 GB of RAM.

Our experimental results are shown in Table 3. The number of nodes of the considered BNs ranges from 56 to 1041. For each Bayesian network, we computed the expected sampling time (EST) for different collections of observed nodes (#obs). Furthermore, Table 3 provides the *average Markov Blanket size*, i.e. the average number of parents, children and children's parents of nodes in the BN [43], as an indicator measuring how independent nodes in the BN are.

Observations were picked at random. Note that the time required by our prototype varies depending on both the number of observed nodes and the actual observations. Thus, there are cases in which we run out of memory although the total number of observations is small.

In order to obtain an understanding of what the EST corresponds to in actual execution times on a real machine, we also performed simulations for the `win95pts` network. More precisely, we generated Java programs from this network analogously to the translation in Sect. 5. This allowed us to approximate that our Java setup can execute $9.714 \cdot 10^6$ steps (in terms of EST) per second.

For the `win95pts` with 17 observations, an EST of $1.11 \cdot 10^{15}$ then corresponds to an expected time of approximately 3.6 *years* in order to obtain a *single* valid sample. We were additionally able to find a case with 13 observed nodes where our tool discovered within 0.32 s an EST that corresponds to approximately 4.3 *million years*. In contrast, exact inference using variable elimination was almost instantaneous. This demonstrates that knowing expected sampling times upfront can indeed be beneficial when selecting an inference method.

[6] http://www.cs.cmu.edu/~fgcozman/Research/InterchangeFormat/.

Table 3. Experimental results. Time is in seconds. MO denotes out of memory.

BN	#obs	Time	EST	#obs	Time	EST	#obs	Time	EST
`hailfinder`	*#nodes: 56, #edges: 66, avg. Markov Blanket: 3.54*								
	0	0.23	$9.500 \cdot 10^1$	5	0.63	$5.016 \cdot 10^5$	9	0.46	$9.048 \cdot 10^6$
`hepar2`	*#nodes: 70, #edges: 123, avg. Markov Blanket: 4.51*								
	0	0.22	$1.310 \cdot 10^2$	1	1.84	$1.579 \cdot 10^2$	2	MO	–
`win95pts`	*#nodes: 76, #edges: 112, avg. Markov Blanket: 5.92*								
	0	0.20	$1.180 \cdot 10^2$	1	0.36	$2.284 \cdot 10^3$	3	0.36	$4.296 \cdot 10^5$
	7	0.91	$1.876 \cdot 10^6$	12	0.42	$3.973 \cdot 10^7$	17	61.73	$1.110 \cdot 10^{15}$
`pathfinder`	*#nodes: 135, #edges: 200, avg. Markov Blanket: 3.04*								
	0	0.37	217	1	0.53	$1.050 \cdot 10^4$	3	31.31	$2.872 \cdot 10^4$
	5	MO	–	7	5.44	∞	7	480.83	∞
`andes`	*#nodes: 223, #edges: 338, avg. Markov Blanket: 5.61*								
	0	0.46	$3.570 \cdot 10^2$	1	MO	–	3	1.66	$5.251 \cdot 10^3$
	5	1.41	$9.862 \cdot 10^3$	7	0.99	$8.904 \cdot 10^4$	9	0.90	$6.637 \cdot 10^5$
`pigs`	*#nodes: 441, #edges: 592, avg. Markov Blanket: 3.66*								
	0	0.57	$7.370 \cdot 10^2$	1	0.74	$2.952 \cdot 10^3$	3	0.88	$2.362 \cdot 10^3$
	5	0.85	$1.260 \cdot 10^5$	7	1.02	$1.511 \cdot 10^6$	8	MO	–
`munin`	*#nodes: 1041, #edges: 1397, avg. Markov Blanket: 3.54*								
	0	1.29	$1.823 \cdot 10^3$	1	1.47	$3.648 \cdot 10^4$	3	1.37	$1.824 \cdot 10^7$
	5	1.43	∞	9	1.79	$1.824 \cdot 10^{16}$	10	65.64	$1.153 \cdot 10^{18}$

7 Conclusion

We presented a syntactic notion of independent and identically distributed probabilistic loops and derived dedicated proof rules to determine exact expected outcomes and runtimes of such loops. These rules do not require any user–supplied information, such as invariants, (super)martingales, etc.

Moreover, we isolated a syntactic fragment of probabilistic programs that allows to compute expected runtimes in a highly automatable fashion. This fragment is non–trivial: We show that all Bayesian networks can be translated into programs within this fragment. Hence, we obtain an automated formal method for computing expected simulation times of Bayesian networks. We implemented this method and successfully applied it to various real–world BNs that stem from, amongst others, medical applications. Remarkably, our tool was capable of proving extremely large expected sampling times within seconds.

There are several directions for future work: For example, there exist subclasses of BNs for which exact inference is in P, e.g. polytrees. Are there analogies for probabilistic programs? Moreover, it would be interesting to consider more complex graphical models, such as recursive BNs [16].

References

1. Abramson, B., Brown, J., Edwards, W., Murphy, A., Winkler, R.L.: Hailfinder: a Bayesian system for forecasting severe weather. Int. J. Forecast. **12**(1), 57–71 (1996)
2. Andreassen, S., Jensen, F.V., Andersen, S.K., Falck, B., Kjærulff, U., Woldbye, M., Sørensen, A., Rosenfalck, A., Jensen, F.: MUNIN: an expert EMG Assistant. In: Computer-Aided Electromyography and Expert Systems, pp. 255–277. Pergamon Press (1989)
3. Batz, K., Kaminski, B.L., Katoen, J., Matheja, C.: How long, O Bayesian network, will I sample thee? arXiv extended version (2018)
4. Bishop, C.: Pattern Recognition and Machine Learning. Springer, New York (2006)
5. Bournez, O., Garnier, F.: Proving positive almost-sure termination. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 323–337. Springer, Heidelberg (2005). https://doi.org/10.1007/978-3-540-32033-3_24
6. Brázdil, T., Kiefer, S., Kucera, A., Vareková, I.H.: Runtime analysis of probabilistic programs with unbounded recursion. J. Comput. Syst. Sci. **81**(1), 288–310 (2015)
7. Celiku, O., McIver, A.: Compositional specification and analysis of cost-based properties in probabilistic programs. In: Fitzgerald, J., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 107–122. Springer, Heidelberg (2005). https://doi.org/10.1007/11526841_9
8. Chakarov, A., Sankaranarayanan, S.: Probabilistic program analysis with martingales. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 511–526. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_34
9. Chatterjee, K., Fu, H., Novotný, P., Hasheminezhad, R.: Algorithmic analysis of qualitative and quantitative termination problems for affine probabilistic programs. In: POPL, pp. 327–342. ACM (2016)
10. Chatterjee, K., Novotný, P., Zikelic, D.: Stochastic invariants for probabilistic termination. In: POPL, pp. 145–160. ACM (2017)
11. Constantinou, A.C., Fenton, N.E., Neil, M.: pi-football: a Bayesian network model for forecasting association football match outcomes. Knowl. Based Syst. **36**, 322–339 (2012)
12. Cooper, G.F.: The computational complexity of probabilistic inference using Bayesian belief networks. Artif. Intell. **42**(2–3), 393–405 (1990)
13. Dagum, P., Luby, M.: Approximating probabilistic inference in Bayesian belief networks is NP-hard. Artif. Intell. **60**(1), 141–153 (1993)
14. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Commun. ACM **18**(8), 453–457 (1975)
15. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall, Upper Saddle River (1976)
16. Etessami, K., Yannakakis, M.: Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. JACM **56**(1), 1:1–1:66 (2009)
17. Fioriti, L.M.F., Hermanns, H.: Probabilistic termination: soundness, completeness, and compositionality. In: POPL, pp. 489–501. ACM (2015)
18. Friedman, N., Linial, M., Nachman, I., Pe'er, D.: Using Bayesian networks to analyze expression data. In: RECOMB, pp. 127–135. ACM (2000)
19. Frohn, F., Naaf, M., Hensel, J., Brockschmidt, M., Giesl, J.: Lower runtime bounds for integer programs. In: Olivetti, N., Tiwari, A. (eds.) IJCAR 2016. LNCS (LNAI), vol. 9706, pp. 550–567. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-40229-1_37

20. Goodman, N.D.: The principles and practice of probabilistic programming. In: POPL, pp. 399–402. ACM (2013)
21. Goodman, N.D., Mansinghka, V.K., Roy, D.M., Bonawitz, K., Tenenbaum, J.B.: Church: A language for generative models. In: UAI, pp. 220–229. AUAI Press (2008)
22. Gordon, A.D., Graepel, T., Rolland, N., Russo, C.V., Borgström, J., Guiver, J.: Tabular: a schema-driven probabilistic programming language. In: POPL, pp. 321–334. ACM (2014)
23. Gordon, A.D., Henzinger, T.A., Nori, A.V., Rajamani, S.K.: Probabilistic programming. In: Future of Software Engineering, pp. 167–181. ACM (2014)
24. Heckerman, D.: A tutorial on learning with Bayesian networks. In: Holmes, D.E., Jain, L.C. (eds.) Innovations in Bayesian Networks. Studies in Computational Intelligence, vol. 156, pp. 33–82. Springer, Heidelberg (2008)
25. Hehner, E.C.R.: A probability perspective. Formal Aspects Comput. **23**(4), 391–419 (2011)
26. Hoffman, M.D., Gelman, A.: The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. J. Mach. Learn. Res. **15**(1), 1593–1623 (2014)
27. Jiang, X., Cooper, G.F.: A Bayesian spatio-temporal method for disease outbreak detection. JAMIA **17**(4), 462–471 (2010)
28. Kaminski, B.L., Katoen, J.-P.: On the hardness of almost–sure termination. In: Italiano, G.F., Pighizzini, G., Sannella, D.T. (eds.) MFCS 2015. LNCS, vol. 9234, pp. 307–318. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-48057-1_24
29. Kaminski, B.L., Katoen, J.: A weakest pre-expectation semantics for mixed-sign expectations. In: LICS (2017)
30. Kaminski, B.L., Katoen, J.-P., Matheja, C., Olmedo, F.: Weakest precondition reasoning for expected run–times of probabilistic programs. In: Thiemann, P. (ed.) ESOP 2016. LNCS, vol. 9632, pp. 364–389. Springer, Heidelberg (2016). https://doi.org/10.1007/978-3-662-49498-1_15
31. Koller, D., Friedman, N.: Probabilistic Graphical Models - Principles and Techniques. MIT Press, Cambridge (2009)
32. Kozen, D.: Semantics of probabilistic programs. J. Comput. Syst. Sci. **22**(3), 328–350 (1981)
33. Kozen, D.: A probabilistic PDL. J. Comput. Syst. Sci. **30**(2), 162–178 (1985)
34. Lassez, J.L., Nguyen, V.L., Sonenberg, L.: Fixed point theorems and semantics: a folk tale. Inf. Process. Lett. **14**(3), 112–116 (1982)
35. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Springer, New York (2004). http://doi.org/10.1007/b138392
36. Minka, T., Winn, J.: Infer.NET (2017). http://infernet.azurewebsites.net/. Accessed Oct 17
37. Minka, T., Winn, J.M.: Gates. In: NIPS, pp. 1073–1080. Curran Associates (2008)
38. Monniaux, D.: An abstract analysis of the probabilistic termination of programs. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 111–126. Springer, Heidelberg (2001). https://doi.org/10.1007/3-540-47764-0_7
39. Neapolitan, R.E., Jiang, X.: Probabilistic Methods for Financial and Marketing Informatics. Morgan Kaufmann, Burlington (2010)
40. Nori, A.V., Hur, C., Rajamani, S.K., Samuel, S.: R2: an efficient MCMC sampler for probabilistic programs. In: AAAI, pp. 2476–2482. AAAI Press (2014)
41. Olmedo, F., Kaminski, B.L., Katoen, J., Matheja, C.: Reasoning about recursive probabilistic programs. In: LICS, pp. 672–681. ACM (2016)

42. Onisko, A., Druzdzel, M.J., Wasyluk, H.: A probabilistic causal model for diagnosis of liver disorders. In: Proceedings of the Seventh International Symposium on Intelligent Information Systems (IIS-98), pp. 379–387 (1998)
43. Pearl, J.: Bayesian networks: a model of self-activated memory for evidential reasoning. In: Proceedings of CogSci, pp. 329–334 (1985)
44. Pfeffer, A.: Figaro: an object-oriented probabilistic programming language. Charles River Analytics Technical Report 137, 96 (2009)
45. Ramanna, S., Jain, L.C., Howlett, R.J.: Emerging Paradigms in Machine Learning. Springer, Heidelberg (2013)
46. Scutari, M.: Bayesian Network Repository (2017). http://www.bnlearn.com
47. Sharir, M., Pnueli, A., Hart, S.: Verification of probabilistic programs. SIAM J. Comput. **13**(2), 292–314 (1984)
48. Wood, F., van de Meent, J., Mansinghka, V.: A new approach to probabilistic programming inference. In: JMLR Workshop and Conference Proceedings, AISTATS, vol. 33, pp. 1024–1032 (2014). JMLR.org
49. Yuan, C., Druzdzel, M.J.: Importance sampling algorithms for Bayesian networks: principles and performance. Math. Comput. Model. **43**(9–10), 1189–1207 (2006)
50. Zweig, G., Russell, S.J.: Speech recognition with dynamic Bayesian networks. In: AAAI/IAAI, pp. 173–180. AAAI Press/The MIT Press (1998)

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.